Journal of Computer Applications
计算机应用,2020,40(9):2577 - 2585
ISSN 1001-9081
CODEN JYIIDU
2020-09-10
http://www.joca.cn

文章编号:1001-9081(2020)09-2577-09
DOI:10.11772/j.issn.1001-9081.2020010077

纯用户态的网络文件系统——RUFS

董豪宇,陈　康*
(清华大学 计算机科学与技术系,北京 100084)
(*通信作者电子邮箱 chenkang@tsinghua.edu.cn)

摘　要:针对在使用高速存储硬件时常规网络文件系统会被软件开销影响整体性能的问题,提出了利用存储性能开发套件(SPDK)搭建文件系统的方法,并在此基础上实现了一个网络文件系统RUFS的原型。该系统通过键值存储模拟文件系统的目录树结构以及对文件系统的元数据进行管理,通过SPDK存储文件的内容。另外,利用远程直接内存访问(RDMA)技术对外提供文件系统服务。RUFS相较于NFS+ext4,在4 KB随机访问上,读写吞吐性能分别提高了202.2%和738.9%,读写平均延迟分别降低了74.4%和97.2%;在4 MB顺序访问上,读写吞吐性能分别提高了153.1%和44.0%。在大部分元数据操作上,RUFS相比NFS+ext4也有显著优势,特别是文件夹创建操作,RUFS的吞吐性能提高了约5 693.8%。该系统能够充分发挥高速网络和高速存储设备的性能优势,为用户提供延时更低、吞吐性能更好的文件系统服务。

关键词:文件系统;远程直接内存访问;存储性能开发套件;用户态系统;固态硬盘
中图分类号:TP302.1　　**文献标志码**:A

RUFS: a pure userspace network file system

DONG Haoyu, CHEN Kang*
(*Department of Computer Science and Technology, Tsinghua University, Beijing* 100084, *China*)

Abstract: The overall performance of traditional network file system is affected by software overhead when using high-speed storage device. Therefore, a method of constructing a file system using SPDK (Storage Performance Development Kit) was proposed, and a prototype of a network file system RUFS (Remote Userspace File System) was realized on this basis. In this system, the directory tree structure of file system was simulated and the metadata of file system were managed by using key-value storage, and the file contents were stored by using SPDK. Besides, RDMA (Remote Direct Memory Access) technology was used to provide file system service to clients. Compared with NFS+ext4, on 4 KB random access, RUFS had the read and write bandwidth performance increased by 202.2% in read and 738.9% respectively, and had the average read and write latency decreased by 74.4% and 97.2% respectively; on 4 MB sequential access, RUFS had the read and write bandwidth performance increased by 153.1% and 44.0% respectively. RUFS had significant advantages over NFS+ext4 on most metadata operations, especially on the operation of folder creation, RUFS had the bandwidth performance increased by about 5 693.8%. File system service with lower latency and higher bandwidth can be provided by this system via making full use of the performance advantages of the high-speed network and high-speed storage device.

Key words: file system; Remote Direct Memory Access (RDMA); Storage Performance Development Kit (SPDK); userspace system; Solid State Disk (SSD)

0　引言

传统的文件系统,例如Ext4[1],都实现在内核态;但内核态的编程需要具备内核相关的知识,有很高的开发门槛。内核态程序的错误常常会影响到整个操作系统稳定运行。如果程序使用到了内核的内部接口,整个程序会变得难以维护和移植。由于这些原因,将文件系统实现在用户态成为了新的趋势。分布式的文件系统,例如GlusterFS(Gluster File System)[2]、GFS(Google File System)[3],由于涉及到复杂的容错策略和网络通信,本身的逻辑很复杂,几乎都实现在用户态。本地文件系统添加某些功能(例如加密[4-5],检查点(checkpoint)设置优化[6]等),也会优先考虑使用FUSE(File system in USErspace)[7]搭建堆栈文件系统,将额外的功能实现在用户态,而不是直接在内核的文件系统上作改动。许多出于研究目的搭建的文件系统[8-10],也都通过FUSE实现在了用户态。

许多用户态的文件系统,存储过程基于本地的文件系统,在存储的过程当中,会发生用户态与内核态的切换。这种切换会引发系统调用、上下文切换、用户态与内核之间内存的拷贝,这些过程为系统带来了额外的软件开销。

新一代的存储硬件——NVMe(Non-Volatile Memory express)固态硬盘(Solid State Drive, SSD),能够提供10 μs以下的延迟和高达3 GB/s的带宽。硬件速度的提高,使得存储

收稿日期:2020-02-04;**修回日期**:2020-03-01;**录用日期**:2020-03-10。　**基金项目**:国家重点研发计划项目(2016YFB1000504)。
作者简介:董豪宇(1994—),男,云南昭通人,硕士研究生,主要研究方向:文件系统、存储技术;　陈康(1976—),男,浙江台州人,副教授,博士,CCF会员,主要研究方向:分布式系统、系统虚拟化、机器学习。

系统的软件开销成为了不可忽视的一部分[11]。如果将整个存储过程迁移到用户态，消除系统因为进入内核而产生的开销，整个系统的性能就可能得到改善。

出于这样的目的，Intel开发了一套高性能的存储性能开发套件（Storage Performance Development Kit，SPDK）[12]。SPDK诞生于2015年，目前学术界已经有了一些基于SPDK的研究[13-15]。由于绕过了内核，并采用了轮询的事件处理方式，相较于内核驱动，SPDK的NVMe驱动能够提供更低且更稳定的延迟。在用户态驱动之上，SPDK还提供了具有不同语义的存储服务，开发者可以在此基础上进行存储系统的开发，而无需关注驱动的实现细节。

在另外一个方面，随着InfiniBand等新硬件的成本下降[16]，以及RoCE（RDMA over Converged Ethernet）[17]技术的成熟，RDMA（Remote Direct Memory Access）技术逐渐在数据中心中普及，学术界也诞生了许多基于RDMA构建的系统[18-21]和相关的研究[22-24]。RDMA技术允许机器在目标机器CPU不参与的情况下，远程地读写目标机器中的内存。相较于传统的TCP/IP网络栈，RDMA技术不仅能提供更低的延迟和更高的带宽[25]，还减少了CPU的开销。由于RDMA协议工作在用户态，使用RDMA进行数据传输，还能避免内核-用户态切换、内存拷贝等过程带来的开销。

当前的用户态文件系统，都依赖于本地文件系统进行实际存储。由于内核-用户态切换的开销，无法充分地利用NVMe SSD的性能。另外一方面，多数文件系统为了实现较高的性能，默认不保证数据实时保存到磁盘介质上。本文希望设计一个纯用户态的网络文件系统，减少存储过程中的软件开销，充分发挥NVMe SSD的硬件性能，并提供同步语义，保证数据实时持久化。同时，利用RDMA进行网络通信，对外提供一个高吞吐和低延迟的文件系统服务。

本文设计并实现了一个基于高速网络与SSD的网络文件系统——RUFS（Remote Userspace File System）。RUFS遵循客户端/服务器端架构，采用RDMA协议进行通信。用户可以利用客户端提供的API，使用由服务器端提供的文件系统服务。服务器端是一个单机的文件系统，元数据管理基于键值存储RocksDB（Rocks DataBase），数据管理基于SPDK Blobstore，所有存储过程都通过SPDK提供的NVMe驱动运行在用户态。通常遵循POSIX（Portable Operating System Interface X）语义的文件系统，只能保证元数据操作的原子性。而RUFS具备同步语义，能够在遵循POSIX语义基础之上，保证已经完成的数据和元数据操作，在服务器掉电之后不丢失，而无需使用fsync进行持久化。

仅使用一块SSD作为数据盘，RUFS：在4KB随机访问上，读、写操作就能获得超过400 MB/s的性能，较默认配置下NFS+ext4的性能提升了202.2%和738.9%；对于4MB顺序访问，RUFS相较于NFS+ext4也有至少40%的性能提升。在元数据性能上，RUFS的文件夹创建性能，相较于NFS+ext4，有5 693.8%的性能提升，其他大部分元数据操作也有显著的性能优势。

本文主要有三个方面的贡献：第一，为如何在用户态完成文件系统所有存储过程给出了详细的方案；第二，在此基础上，实现了一个网络文件系统原型RUFS，并报告了数据和元数据性能；第三，改进了BlobFS的缓存策略，使得工作在BlobFS之上的键值存储的读性能有了非常明显的提升，也间接提升了RUFS的元数据性能。

1 相关研究

1.1 基于键值存储的文件系统元数据管理

键值存储是一种NoSQL存储，一般基于LSM Tree（Log Structured Merged Tree）[26]构建，提供有序键到任意长度值的持久化存储和查询。通过将随机写入转化为顺序写入，这种键值存储通常能够取得更好的性能。

2013年，Ren等[8]提出了TableFS（Table File System）。TableFS利用LevelDB（Level DataBase）[27]构建了一个文件系统元数据模块，以键值对的键描述父节点到子节点的关系，并将文件的元数据作为键值对的值存在LevelDB中。在此基础上，利用Ext4作为对象存储，为TableFS提供数据的存储服务。为了减少对下层Ext4的访问，TableFS还将小于4 KB的文件也放在了LevelDB当中。

在此基础上，2014年，Ren等[28]提出了TableFS的分布式版本——IndexFS（Index File System），并在此基础上做了一些相关的工作[29-30]。2017年，Li等[31]提出了LocoFS（Loco File System）。LocoFS改进了用键值存储模拟目录树的方式，减少了元数据操作需要的网络请求数量，提高了元数据操作的性能。

1.2 SPDK技术

存储服务的性能由软件与硬件共同决定。对于传统存储硬件（如机械硬盘），由于硬件性能较差，软件上的开销只占整个存储服务开销的一小部分。但随着NVMe SSD的出现，相当一部分存储硬件，例如Z-SSD、Optane SSD等，已经能够提供小于10 μs的延迟和高达3 GB/s的带宽[11]。硬件性能的大幅提高，使得软件栈的开销成为了整个存储服务开销中不可忽视的一部分。

为了充分利用NVMe SSD的性能，减少存储过程中的软件开销，Intel开发了SPDK。SPDK提供了一个纯用户态的NVMe驱动，消除了内核与系统中断带来的开销。SPDK提供的NVMe驱动采用了无锁的实现，支持多线程同时提交I/O请求。

根据SPDK团队的论文[12]，对于NVMe SSD（实验所用的SSD型号为Intel P3700，容量为800 GB）的4 KB随机访问。SPDK的用户态NVMe驱动，能用1块SSD提供450 kIOPS（Input/output Operations Per Second）的访问性能，略高于Linux内核中的NVMe驱动。得益于无锁的实现方式，SPDK提供的性能，能够随着SSD的增多而线性增加，用8块SSD提供约3 600 kIOPS的访问性能。而增加SSD数量，对内核驱动提供的性能没有提高。在4 KB随机读的延迟上，SPDK能够将内核驱动造成的软件开销，降低约90%。

SPDK为用户提供了一套事件驱动的编程框架。在这套框架中，每个线程相互独立，通过消息传递的方式进行线程间同步，线程间不共享任何资源。这种设计消除了资源共享带来的数据竞争，使框架具有良好的可扩展性。这个编程框架定义了3个重要的概念，分别是reactor、event和poller。reactor是一个常驻的线程，持有一个无锁的消息队列；event代表一个任务，可以通过reactor的消息队列在线程间传递；poller与event类似，也是一种任务，但需要注册在一个reactor上，reactor会周期性地调用已注册的poller。用户可以使用poller在用户态模拟系统中断。

Blobstore和BlobFS（Blob File System）是SPDK提供的两个存储服务，前者提供对象（blob）存储的语义，后者提供一个简易的文件系统，用户可以在此基础上搭建存储应用。Blobstore中最基础的存储单元被称为page，每个page 4 KB大小。Blobstore可以保证每个page写入的原子性。根据用户配置，Blobstore会将连续的多个page组织在一起（通常大小为1 MB），这一段连续

的空间被称为一个cluster,而blob则是一个cluster的链表。用户可以在blob上进行随机、并发、无缓存的读写,还可以将键值对以元数据的形式存储在blob当中,但元数据需要用户自己手动调用sync md(同步元数据)操作才能持久化。

BlobFS是在Blobstore的基础上构建的一个简易的文件系统。每个文件都对应着Blobstore中的一个blob,文件的名字和长度,都以键值对的形式存储在blob的元数据当中。BlobFS只能支持创建根目录下的文件,不支持文件夹功能,不支持随机位置的写入,只支持增量写。当前BlobFS已经能够作为键值存储系统的存储引擎,但由于不支持随机位置的写入,仍然不适合用于管理文件数据。BlobFS当中还实现了一个简单的缓存模块,可以为文件的顺序读和增量写带来一定的性能提升。

1.3　基于RDMA的RPC技术

RDMA是指一种允许处理器直接读写远程计算机内存的技术。相较于传统网络,RDMA能够提供极低的延时和很高的带宽。最新商用的RDMA网卡可以提供低至600 ns的延时和每端口高达200 Gb/s的带宽[25]。RDMA编程一般使用verbs API,需要开发者自己控制网络连接、任务轮询等细节,编程复杂度较高。

RPC(Remote Procedure Call)技术[32],是一种允许本地机器透明地调用远端函数或者过程的技术。RPC技术向用户隐藏了数据的发送、接收、序列化、反序列化等细节,大大降低了编程复杂度。Mercury是面向超算领域的RPC框架,于2013年由Soumagne等[33]提出。Mercury包含了一个网络抽象层,可以通过不同的通信插件,在不同网络硬件下进行数据传输。当前,Mercury采用了OFI(Open Fabric Interface)[34]作为支持RDMA传输的通信插件。Mercury在常规的RPC接口之外,还提供了一组块(bulk)传输接口。Bulk接口能够充分利用RDMA单边通信的性能,消除不必要的内存拷贝。用户可以把一块内存注册为一个bulk,并将bulk句柄发送给其他机器,其他机器就能通过bulk句柄远程地读写被注册的内存。在Mercury的基础上,Intel正在开发一套支持组通信的RPC框架,CaRT(Collective and RPC Transport),作为其在新的存储系统DAOS(Distributed Asynchronous Object Storage)[35-36]中的传输层。CaRT不仅支持传统的点对点RPC通信,还能支持RPC的组播。

2　系统架构与设计

本章主要介绍了RUFS的设计与实现,包括元数据管理的策略、数据管理的策略、保证元数据与数据的一致性策略。

2.1　系统架构

RUFS是一个纯用户态的文件系统,采用客户端/服务器端架构,服务器端是一个单机系统,可以同时支持多个客户端。服务器端与客户端通过CaRT进行通信。

RUFS客户端为用户提供了一套类POSIX语义的、并发安全的文件系统操作API(RUFS-API),支持的操作包括:access、mkdir、rmdir、stat、rename、opendir、readdir、closedir、open、creat、close、ftruncate、unlink、pread、pwrite、read、write,支持文件和文件夹操作。当用户调用客户端API时,客户端会将请求通过RPC的形式发送到RUFS服务器端,并等待请求返回。

RUFS服务器端实现了一个纯用户态文件系统(RUFS-server)。系统需要至少两块SSD才能工作,其中一块用来建立BlobFS实例,用来支持RocksDB[37]的数据存储。RUFS将利用RocksDB对元数据进行存储和管理。剩下的每块SSD都会创建一个单独的Blobstore实例,用来存储数据,其中的每个blob都包含着一个文件的数据。RUFS能利用多个SSD来提高服务器端的文件读写的吞吐能力。在RUFS服务启动时,系统会为每一块用于存储数据的SSD建立一个Blobstore实例,同时,启动一定数量的reactor线程,负责处理读写请求。reactor线程的数量可以自行配置,但不能超过Blobstore的实例数量,每个Blobstore实例受一个固定的reactor线程的管理。

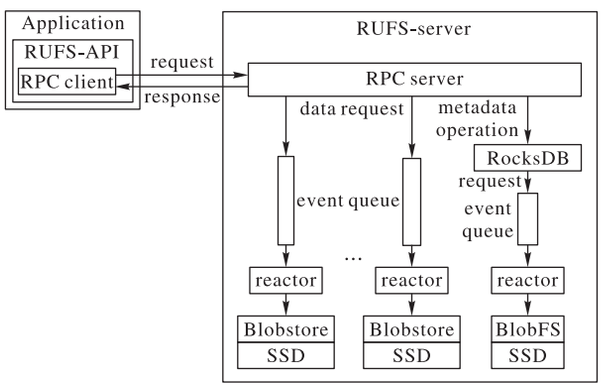


图1　RUFS架构

Fig. 1　RUFS architecture

2.2　元数据管理

2.2.1　基于键值存储与Blobstore的元数据协同管理

文件系统的元数据通常组织为目录树。目录树的节点存储了文件的元数据,每一个节点都有一个唯一的编号(inode number),目录树的边代表目录对下一级节点的包含关系。RUFS利用键值存储模拟目录树,同样模拟了目录树的节点和边,并为每个节点赋予了一个唯一的UUID(Universally Unique IDentifier)。目录树的节点和边用不同的键值对模拟,前者称为N型(node)键值对,后者称为E型(edge)键值对。两者在键值存储中,有不同的前缀,N型键值对模拟节点,键由前缀、节点UUID拼接而成,值包含了该节点的一部分元数据(记为Meta-N),E型键值对模拟父节点到子节点的边,键由前缀、父节点UUID、子节点文件名拼接而成,值中包含子节点UUID和子节点的一部分元数据(记为Meta-E)。基于键值存储的键的有序性,拥有相同父节点的E型键值对会聚合到一起,这方便了readdir的实现。RUFS可以将readdir对子节点的遍历,转化为RocksDB对键值对的遍历。

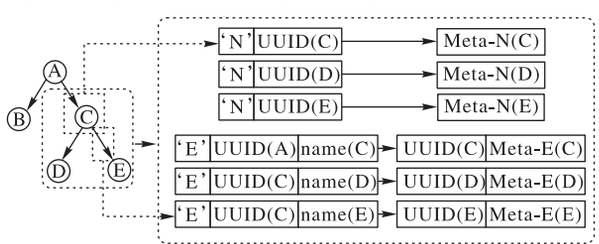


图2　目录树与键值对的对应关系

Fig. 2　Relationship between directory tree and key-value pairs

在RUFS中,一个文件/文件夹的元数据包括:mode(其中包含了节点类型、权限信息)、gid、uid、atime、ctime、mtime。对于文件,还包括文件的长度、文件对应的blob的相关信息。许多元数据操作的接口,都是基于路径名的(例如creat、rmdir等),系统需要从根节点开始,通过文件名和E型键值对,顺着目录树的树边逐层往下查找节点,直到找到路径名对应的节点,再做相应的操作。在查找目标节点的过程中,根据POSIX语义的要求,系统同时要判断操作对节点是否有访问权限。这需要读取节点元数据中的mode、gid和uid。为了消除在权限判断过程中,对N型键值对的额外访问,RUFS将mode、gid

和uid划分到了E型键值对中。图3展示了节点元数据是如何存储在不同的键值对中的。

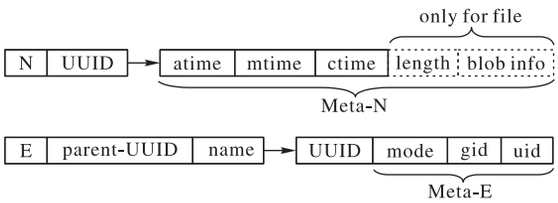


图3 在键值对中存储元数据的方式

Fig. 3 Method of storing metadata in key-value pairs

根据POSIX语义的要求,文件在被进行读写时,文件的时间戳需要被相应地改变,文件的长度也可能发生变化。如果要将这些改动同步到RocksDB当中,当系统需要同时处理大量的读写请求时,RocksDB的写入性能就会成为整个系统的瓶颈。因此,在RUFS中,文件的长度和时间戳还会存储在对应的blob的元数据中。当文件被读写时,文件长度和时间戳的变化只会存储到blob的元数据中,当文件被关闭时,长度和时间戳才会被同步回RocksDB中。

2.2.2 元数据操作的原子性和并发安全性

某些元数据操作(例如rename)需要对目录树进行多次改动,为了保证操作的原子性,RUFS中所有可能涉及目录树变化,或是在操作过程中默认目录树不发生结构变化的操作,都利用了RocksDB事务来保证元数据操作的原子性。

RUFS服务器端作为一个多线程的系统,能够并发地更改目录树的结构,如果不进行并发控制,就会产生错误。而单纯使用RocksDB事务,无法避免这样的错误。图4展示了一种出错的情况。在目录树为初始状态时,系统同时收到了creat操作和rmdir操作,由于RocksDB事务只处理写-写操作的冲突,因此两个元数据操作事务得以并发地执行,并进行了事务提交,结果造成了creat操作创建了一个空悬的节点。为了解决这一问题,RUFS利用了RocksDB事务中的get_for_update操作。这一操作会促使RocksDB为目标键值对加上一个读写锁,通过为目录树上的节点和边上读写锁,就能避免元数据的并发操作破坏目录树结构。在加锁的顺序上,RUFS总是遵循这样的规则:对于两个待加锁的节点A和B,若两者在目录树中深度不同,那么RUFS会从较浅的节点到较深的节点加锁。若两者在目录树中的深度相同,RUFS会从UUID较小者到UUID较大者加锁。RUFS通过有顺序的加锁,来避免死锁问题。

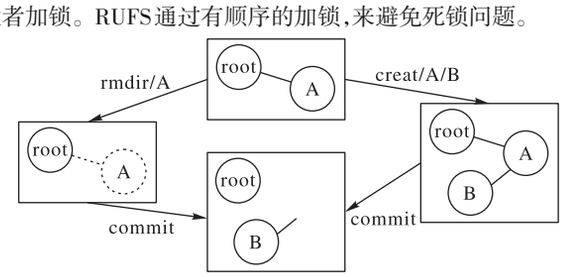


图4 并发的元数据操作导致的错误

Fig. 4 Error caused by concurrent metadata operations

2.3 数据管理

SPDK提供3个存储服务,分别是BDev(Block Device)、Blobstore和BlobFS。其中:BDev只提供块设备的语义,过于简单,不适合用作管理文件数据;BlobFS不支持对文件的随机写入;而Blobstore则能提供对象存储的语义,提供针对blob的创建、删除、随机读写、长度变更等操作。RUFS容易将针对文件内容的操作,映射到Blobstore中针对blob的操作。因此,RUFS选择将数据存储在Blobstore中。

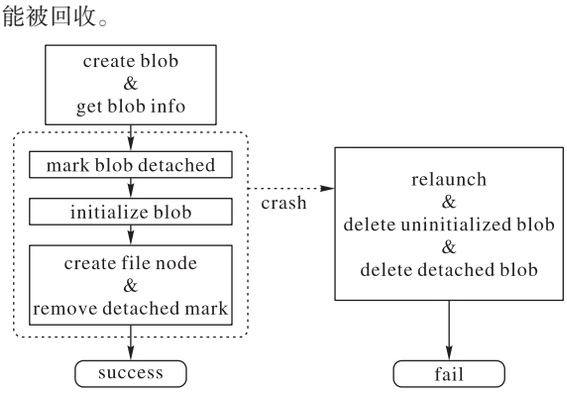
每创建一个文件,RUFS就在Blobstore中创建一个相应的blob。目录树中的文件节点与Blobstore中的blob一一对应。blob的位置信息(blob所属的Blobstore和blob ID),会成为文件元数据的一部分,存储在RocksDB当中。每一个Blobstore实例都由一个固定的reactor管理,对Blobstore的任何操作,包括blob的创建、删除、读写,都需要提交给对应的reactor,由reactor完成。

2.3.1 元数据与数据的一致性策略

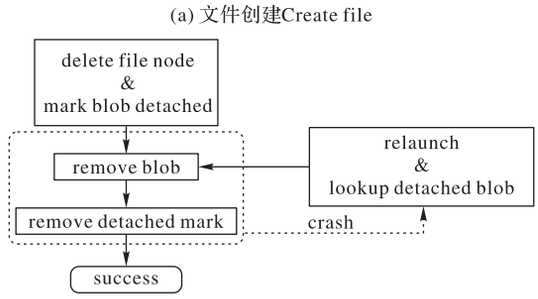
当用户创建或删除一个文件时,RUFS不仅需要改变目录树的结构,还需要在Blobstore上创建或者删除相应的blob,维持文件节点与blob的一一对应。宕机会导致文件的创建或删除执行不完整,破坏文件节点与blob一一对应的关系。如果产生了游离的blob(没有对应文件节点的blob),则造成存储空间的泄漏,如果文件节点没有对应的blob,则意味着数据丢失。

RUFS利用了一种基于blob标记的手段来解决这个问题。基本的思路是,将没有和元数据建立联系的blob标记为已解耦(detached),并将位置信息记录到RocksDB中,这样即使服务器宕机,重启RUFS之后,系统也能够回收这些blob。

图5(a)展示了文件的creat过程,新创建的blob会被标记为detached,并记录在RocksDB当中,只有在blob元数据设置成功,并且将位置信息记录在目录树上后,RocksDB中的detached记录才会被删除。如果因为宕机导致操作没有完全执行,RUFS在重新启动时,通过检查blob的元数据和RocksDB中的记录,就能回收游离的blob。Detached记录的删除过程与目录树的操作处于同一个RocksDB事务中,能保证creat成功后,被创建出的blob不会被错误地回收。图5(b)展示了文件的unlink操作,标记blob为detached的过程会和删除文件节点的过程放在同一个RocksDB事务中。这能保证只要元数据的删除操作成功,即使出现意外宕机,游离的blob也总能被回收。



(a) 文件创建Create file



(b) 文件删除Unlink file

图5 创建和删除文件的流程

Fig. 5 Processes of file creation and deletion

2.3.2　句柄与读写状态管理

根据POSIX标准的要求,open、creat、opendir等操作,需要向调用者返回一个句柄。通过句柄,用户可以进一步地对文件或文件夹进行其他操作,而不用再进行从路径到文件节点的搜索和权限判断。在RUFS中,通过句柄,用户可以读写文件(read、write、pread、pwrite),遍历文件夹下的子项目(readdir)。

RUFS的句柄包含两个字段:一个字段是被打开节点的UUID,用来标示被打开的节点;另一个字段是一个唯一的64位无符号数(fd ID),用来标示打开同一个文件的不同句柄。通过句柄,RUFS-server能够查找当前句柄对应的读写状态。

RUFS-server采用了图6中的数据结构来管理句柄的读写状态,这个数据结构用了一个以UUID为键的哈系表,来维持被打开文件/文件夹的内存节点。内存节点中包含了对其进行操作所需要的数据;文件内存节点中包含了文件对应的blob的位置信息,缓存了文件的长度和时间戳;文件夹内存节点,存储了以该文件夹为父节点的E型键值对的键的公共前缀(前缀E+文件夹UUID),这个前缀用来在遍历E型键值对时,判断被访问的键值对是否指向该文件夹的子节点。每个内存节点中包含了一条链表,链表上的每一个元素,都记录了某个句柄对应的读写状态,如果句柄属于一个文件,那么读写状态就是当前偏移(offset)的位置,如果句柄属于一个文件夹,那么读写状态就是readdir操作所需的RocksDB迭代器。利用图6中的数据结构,RUFS还能通过哈希表快速地判断某个节点是否被打开,阻止被打开的节点被删除。

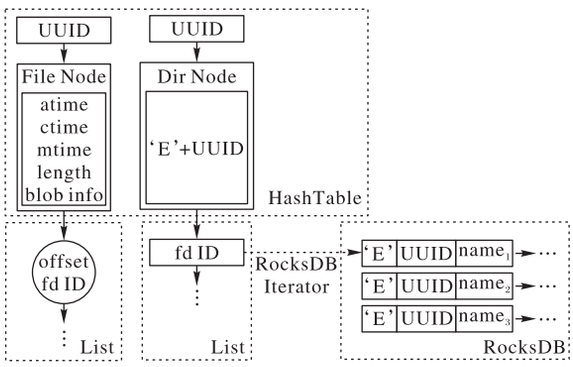


图6　RUFS对句柄的管理

Fig. 6　Management of handles in RUFS

3　系统实现与优化

3.1　网络传输优化

RUFS采用了CaRT作为客户端与服务器端通信的手段。CaRT为用户提供了RPC接口和bulk接口,bulk接口能够充分利用RDMA的单边通信性能,避免不必要的内存拷贝。元数据操作需要传输的数据量通常很少,因此RUFS只使用CaRT提供的RPC接口来发送元数据操作。但读写操作,可能需要传输较多的数据,为了充分利用RDMA的单边通信原语,提高传输性能,RUFS利用bulk接口来传输读写缓冲区中的内容。

但使用bulk接口,需要用户自己申请内存,并将内存注册为一个块(bulk)。注册bulk非常耗时,将一块仅1 B的内存注册为bulk,需要耗费大约60 μs,如果在客户端和服务器端都进行内存的注册,一次通信会产生额外的120 μs的开销,随着被注册内存的增大,耗时还会增大。图7展示了发送不同大小的负载时复用bulk(记为reuse-bulk)和每次注册新的bulk(记为register-bulk)在传输延迟上的性能差距。

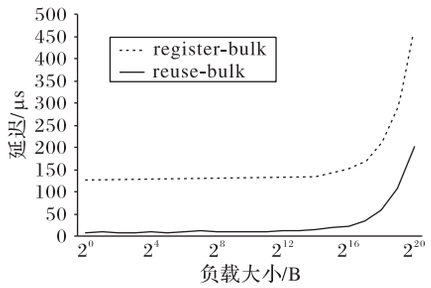


图7　不同负载下bulk传输的延迟

Fig. 7　Bulk transfer latency with different payload sizes

为了解决这一问题,RUFS设计了一个内存池,Bulk-Mempool。Bulk-Mempool会提前将一些大块的内存注册为一个bulk,并在这块内存上进行进一步的分配。在读写操作的过程中,服务器和客户端用到的读写缓冲区就从这个内存池中分配,这就消除了RUFS在每次读写操作时,将读写缓冲区所在的内存注册为bulk而带来的开销。

Bulk-Mempool并不是一个单一策略的内存池,而是由两个不同策略的内存池BM-small和BM-mid组成的。BM-small实现比较简单,分配开销较小,只分配4 KB大小的内存;BM-mid内部实现了一个buddy system内存池,开销相对较大。小文件读写通常触发小于等于4 KB的内存分配请求,此时由开销较小的BM-small进行内存分配,能够保证小文件读写的性能;大于4 KB、小于等于4 MB的内存分配请求,则由BM-mid负责。BM-mid能够分配不同大小的内存,提高内存的利用率。

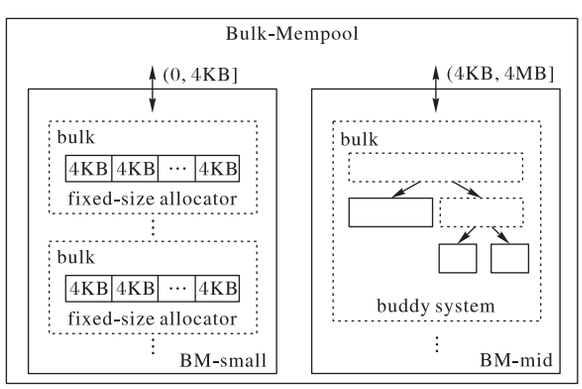


图8　Bulk-Mempool的架构

Fig. 8　Architecture of Bulk-Mempool

Bulk-Mempool没有对大于4 MB的内存分配进行优化,是因为以4 MB为单位的数据传输,已经能够充分地利用InfiniBand的高带宽,使数据传输不会成为整个系统的瓶颈,本文的实验也说明了这一点(见4.3节)。如果用户需要传输大文件,将每次读写请求分割为4 MB大小即可。

Bulk-Mempool提供get和put接口。通过get接口,调用者能够获得一个胖指针,其中包括了指向被分配内存的指针、被分配内存的大小、被分配内存在bulk中的偏移,以及bulk句柄。前两者使得调用者可以在本地读写分配到的内存,后两者使得远端机器能够正确在分配到的内存上进行读写。

3.2　读写吞吐能力优化

按照POSIX语义的要求,文件的读写会导致文件的大小

和时间戳发生变化，导致元数据的改动。这在Blobstore中就表现为需要通过sync md（同步元数据）操作同步blob的元数据，但sync md操作非常耗时，甚至超过了一次4 KB的读或写。如果每次文件读写都要更新时间戳或是文件长度，频繁触发sync md操作，会让整个系统的吞吐能力受到极大的影响。为此，RUFS提供了两个优化策略，以减小sync md的触发频率。

第一个策略是，提供了ftruncate操作，并鼓励用户尽可能在写入数据前，将文件扩展到合适的大小，这样能够避免写入操作“撑大”文件大小，进而触发sync md。另一个策略是，放宽对时间戳更新的要求，每次进行读写时，将系统当前的时间，与文件当前的时间戳进行比较，只有文件需要更新的时间戳，与当前的时间相差多于5 ms时，才选择更新时间戳。考虑到系统本身就存在着时间上的误差，这样的放松策略是可以接受的。

3.3 可靠元数据性能优化

RocksDB会将写入操作记录到日志里，但并不会立刻将日志写入到磁盘中。在内存中缓存一定数量的日志之后，RocksDB才会一次性地将所有内存中的日志写到硬盘上。这个特性被称作“组提交（group commit）”，组提交特性显著地减少了向磁盘写入数据的次数，对RocksDB的写入性能有很大的提升。但同时，由于写入的数据不能被及时持久化，服务器断电就可能导致元数据操作的丢失。考虑到Blobstore会保证数据持久化后再返回，为了使元数据与数据保持一致，提供同步的文件系统语义，RUFS也需要保证元数据操作返回后，就已经持久化到了硬盘上。

为了提供这样的保证，RUFS打开了RocksDB的同步模式。同步模式下，每次写入操作后，RocksDB都会调用fsync保证日志写入硬盘，使数据在宕机后不丢失。然而，本地文件系统的fsync性能很差，这导致了RocksDB同步模式下的写入性能也很差，降低了RUFS整体的元数据性能。为了解决这个问题，RUFS采用了由SPDK团队修改并开源的RocksDB[38]。这个版本的RocksDB将底层的存储环境更换为了BlobFS。BlobFS有很好的同步写入性能，能够显著提升RocksDB在同步模式下的写入性能。

RocksDB的读操作触发的都是文件的随机读，而BlobFS当前仅针对顺序读进行缓存。缓存的缺失使RocksDB的读性能变得很差。为了解决这个问题，在BlobFS中添加了一个支持缓存的随机读方法，当RocksDB调用这个方法进行读操作时，BlobFS会预取所需数据所在的一个256 KB的数据块，并缓存在内存当中。利用缓存，相比以文件系统作为存储环境，RocksDB在BlobFS上的写性能能得到显著提升，且保持读性能基本相同。

3.4 统一的SPDK环境管理模块

RUFS利用一个或多个Blobstore管理数据，利用由SPDK团队提供的RocksDB管理元数据。这两者都需要工作在SPDK环境下。当前，由SPDK团队提供的RocksDB，会在内部自行启动一个SPDK环境。由于一个进程只能启动一个SPDK环境，因此RocksDB启动的SPDK环境，会和RUFS启动的SPDK环境产生冲突，导致整个系统启动失败。

为了解决这一问题，RUFS去掉了RocksDB中启动SPDK环境的功能，并将这部分功能整合到了RUFS中，再加上对Blobstore依赖的SPDK环境的管理功能，形成了统一的SPDK环境管理模块（SPDK-env-mod）。系统启动时，SPDK-env-mod会初始化SPDK环境，同时创建BlobFS。在RocksDB初始化时，SPDK-env-mod会将BlobFS暴露给RocksDB，使RocksDB顺利在BlobFS上初始化和运行。

除了解决SPDK环境冲突的问题，SPDK-env-mod还方便了系统管理员对Blobstore的管理。SPDK-env-mod提供了一个配置文件，系统管理员可以通过该配置文件，指定用来管理数据的SSD，以及用于管理数据的reactor线程的数量。系统启动后，SPDK-env-mod会根据配置文件，在每块用于管理数据的SSD上，建立Blobstore实例。同时，根据配置文件，启动一定数量的reactor线程，并按照平均分配的原则，将Blobstore绑定到不同的reactor线程上。

除此之外，SPDK-env-mod会给每一个Blobstore赋予一个从0开始的、单调递增的唯一编号，同时在内存中维持一个计数器，每次系统需要创建一个blob时，就将计数器的值对Blobstore的数量取模，以此选出一个Blobstore实例，在这个实例上创建blob，并将计数器原子地加1。由于Blobstore实例与用来管理数据的SSD一一对应，这样的分配方案，可以保证blob均匀地分布在各个SSD上。

4 测试与评估

本章将评估RUFS在元数据、读写延迟和读写吞吐方面的性能。RUFS的总体性能会和NFS+ext4进行比较；而RUFS-server的性能会和ext4进行比较。本章还会讨论SPDK对元数据的加速效果和多SSD对吞吐性能的提升。

4.1 测试配置

所有的测试都在两台服务器上进行，其中一台作为RUFS的服务器，另一台作为RUFS的客户端。RUFS客户端装配了2块6核CPU，128 GB内存；RUFS服务器端装配了4块12核CPU、768 GB内存、8块容量为512 GB的NVMe SSD。两台服务器通过56 Gb/s带宽的InfiniBand网卡相连。表1是测试环境的具体参数。

在所有测试中，NFS与ext4均采用默认配置，ext4建立在服务器端，使用1块SSD，利用NFS挂载到客户端。RUFS使用2块SSD，分别用来管理数据和元数据，使用16个RPC处理线程，1个reactor线程。客户端利用RUFS客户端提供的API访问RUFS服务器。

表1 测试环境设置

Tab. 1 Testing environment configuration

硬件配置	CPU	内存/GB	硬盘	网络
客户端	2×6核	128	—	56 Gb/s
服务器端	4×12核	768	8×512 GB SSD	InfiniBand

4.2 元数据性能

本文采用mdtest[39]对元数据性能进行测试，用每秒的操作数量（Operations Per Second，OPS）衡量性能。该测试对比了NFS+ext4与RUFS整体的元数据性能。在元数据性能测试中，客户端使用8个mdtest进程，文件节点的最大深度为4，文件/文件夹总数大约为50万。如果测试对象为RUFS，需要将mdtest中的文件系统函数换成RUFS-API。

4.2.1 需要关注的元数据操作

在本节的测试中，主要关注如下的元数据操作：D-creat、D-stat、D-remove、F-creat、F-stat、F-read和F-remove，表2展示

了它们的意义和在过程中会触发的操作。

表2　元数据操作和它们的属性和含义

Tab. 2　Metadata operations and their attributions and meanings

操作名称	触发 Blobstore 操作	触发 RocksDB 写操作	含义
D-creat	否	是	创建文件夹
D-stat	否	否	获取文件夹元数据
D-remove	否	是	删除文件夹
F-creat	是	是	创建文件
F-stat	否	否	获取文件元数据
F-read	是	是	打开文件后关闭文件
F-remove	是	是	删除文件

4.2.2　测试结果

从图9来看:RUFS在F-creat和F-remove两个操作上,与NFS+ext4的性能大致相同;在其他元数据操作上,RUFS都具有显著的优势,取得了至少70%的提升;特别对于D-creat操作,RUFS相对于NFS+ext4有大约5 693.8%的性能提升。横向对比RUFS各个元数据操作的性能,F-creat和F-remove由于需要在Blobstore上进行多次操作,因此性能显著低于其他元数据操作。F-read操作包含了一次open操作和一次close操作,且需要访问Blobstore,因此性能也同样较差。

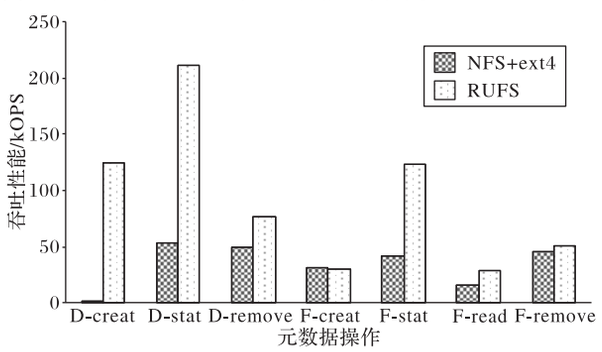


图9　RUFS与NFS+ext4元数据性能的比较

Fig. 9　Metadata performance comparison of RUFS and NFS+ext4

4.2.3　SPDK为元数据带来的性能提升

为了达到同步语义,RUFS在使用RocksDB时会打开同步模式,这会导致RocksDB的写入性能大幅下降。SPDK能够为存储应用带来更低的延迟、更高的吞吐性能。通过将RocksDB的存储环境替换为优化后的BlobFS,RocksDB的同步写性能有了很大的提升,并且读性能没有受到影响。本节将展示BlobFS对元数据性能的影响。

图10展示了RUFS元数据性能在RocksDB在不同配置(同步模式或组提交模式,分别记为sync和group-commit)、不同存储环境下(文件系统或BlobFS,分别记为fs和SPDK)的结果。从非同步模式切换为同步模式,无论存储环境是BlobFS还是文件系统,涉及到RocksDB写入的元数据操作,都会有明显的性能下降。在BlobFS环境下,creat操作性能损耗最大,大约为38.8%,但由于原本性能很好,因此性能依然可以接受。存储环境为本地文件系统时,元数据操作的性能损耗变得不可接受,性能损耗最多的元数据操作依然是creat,损耗比例高达98.7%,基本处于不可用的状态。其他元数据操作,除了D-stat与F-stat不发生RocksDB写入,不受同步模式的影响,其他操作的OPS都小于2 500。

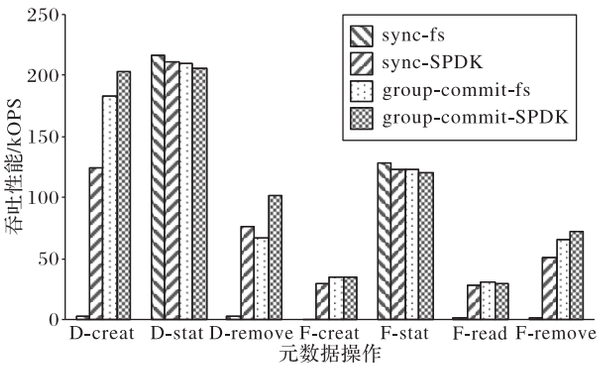


图10　SPDK对元数据操作的性能的影响

Fig. 10　Impact of SPDK on metadata operation performance

4.3　数据性能

本节将讨论RUFS、NFS-ext4、RUFS-server和ext4在4 KB随机读写延迟、4 KB随机读写吞吐、4 MB顺序读写吞吐几个场景上的性能。由于当前RUFS还没有加入对缓存的支持,因此在对ext4进行测量时,尽量消除了缓存对ext4的影响。ext4的写入包括两个项目:ext4-direct和ext4-sync。前者在打开文件时,使用了O_DIRECT选项,避免数据写入到缓存;后者在打开文件时使用了O_SYNC选项,保证写入数据能够持久化到硬盘。需要说明的是,由于O_SYNC选项不影响读操作,因此在测试读性能时,ext4-sync与ext4-direct会使用同一个数据。RUFS-server仍然使用16个RPC处理线程,用1块SSD管理元数据,1块SSD管理数据。

4.3.1　延迟

测量了RUFS-server、ext4-sync、ext4-direct、RUFS、NFS+ext4的4 KB随机读写的延迟,图11展示了测试结果。从结果上来看,RUFS-server的读延迟,大约只有ext4的20%。在网络环境下,RUFS总体的读延迟,只有NFS+ext4的26%左右。而对于本地写性能,RUFS-server仅略快于ext4-direct,但要注意,ext4-direct并不保证操作返回时,能将数据持久化在硬盘上。提供这一保证的ext4-sync,写延迟则是RUFS-server的近60倍,在网络环境下,RUFS总体的写延迟也远远小过NFS+ext4。

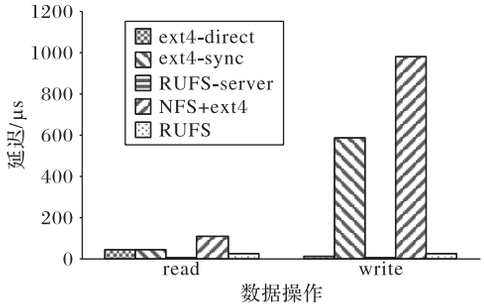


图11　RUFS与NFS+ext4关于4 KB随机访问的延迟

Fig. 11　4 KB random access latency of RUFS and NFS+ext4

4.3.2　吞吐

吞吐性能测试包括了4 KB随机读写和4 MB顺序读写两个项目。图12展示了4 KB随机读写吞吐性能的结果。这个结果和延迟测试类似,RUFS-server在读写性能上,都远远地超过了ext4-sync,同时略强于不提供持久化保证的ext4-direct。总体性能上,RUFS读性能是NFS+ext4的3倍以上,写

性能是NFS+ext4的8倍以上。

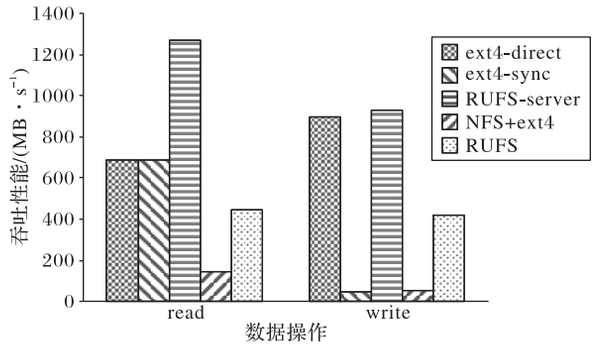


图12 NFS+ext4与RUFS关于4 KB随机访问的吞吐性能

Fig. 12 4 KB random access bandwidth of NFS+ext4 and RUFS

在4 MB的顺序读写上，ext4与RUFS的差距就相对小了一些。没有网络参与时，无论是读还是写，RUFS-server均快于ext4，但性能提升不超过30%。但值得注意的是，在大文件的顺序读写中，RUFS的总体性能与RUFS-server的吞吐性能几乎持平，这意味着网络传输提供了足够高的带宽，没有成为整个系统的瓶颈。

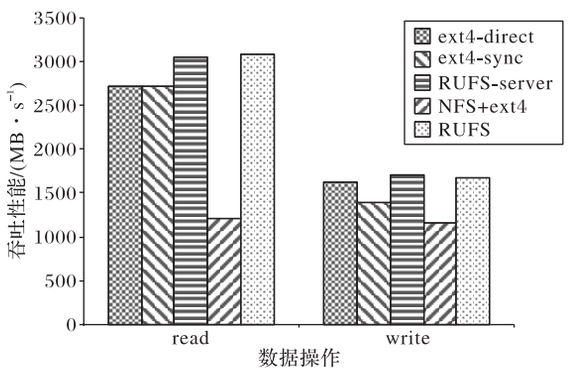


图13 RUFS与NFS+ext4关于4 MB顺序访问的吞吐性能

Fig. 13 4 MB sequential access bandwidth of RUFS and NFS+ext4

4.3.3 多SSD带来的性能提升

RUFS-server默认只用1个SSD管理数据，因此也只使用1个reactor线程管理读写请求。如果使用多个SSD管理数据，RUFS-server就能启动多个reactor处理读写请求，这能够提升RUFS-server的吞吐性能。图14展示了RUFS-server在多块SSD下吞吐性能的提升。当使用6块SSD管理数据时，通过将文件分散到各块SSD，并用6个reactor同时处理读写请求，RUFS-server的吞吐性能能获得246%到450%的提升。

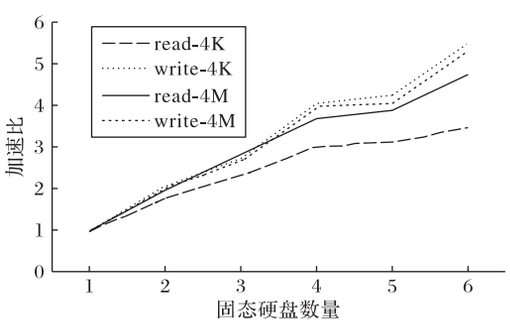


图14 多SSD为RUFS-server带来的加速比

Fig. 14 Speedup ratio brought by multi-SSD on RUFS-server

5 结语

本文设计并实现了一个基于高速网络和NVMe SSD的用户态网络文件系统，RUFS。RUFS利用RocksDB管理元数据，利用Blobstore管理数据，使用RDMA技术对外提供服务。RUFS充分地利用了NVMe SSD的性能，所有的存储过程都通过SPDK提供的NVMe驱动运行在用户态。RUFS在随机读写、顺序读写和元数据性能上，相较于NFS+ext4都有十分明显的优势。除此之外，RUFS还具备同步语义，能够保证用户请求返回后，数据就已经被持久化到硬盘当中。

通过RUFS的开发和测试，也充分证明了SPDK在存储领域的潜力，尤其是保证数据可靠写入、并持久化在硬盘的性能，明显地好于本地文件系统。因此SPDK也十分适合于开发对存储持久性要求较高的应用，例如关系型数据库的存储引擎。

参考文献 (References)

[1] CAO M, BHATTACHARYA S, TSO T. Ext4: the next generation of Ext2/3 filesystem[C/OL]// Proceedings of the 2007 Linux Storage Filesystem Workshop. Berkeley: USENIX, 2007 [2020-01-10]. https://www.usenix.org/legacy/event/lsf07/tech/cao_m.pdf.

[2] BOYER E B, BROOMFIELD M C, PERROTTI T A. GlusterFS one storage server to rule them all[R]. Los Alamos, NM: Los Alamos National Lab, 2012.

[3] GHEMAWAT S, GOBIOFF H, LEUNG S T. The Google file system [C]// Proceedings of the 19th ACM Symposium on Operating Systems Principles. New York: ACM, 2003: 29-43.

[4] HALCROW M A. eCryptfs: an enterprise-class encrypted filesystem for Linux [C]// Proceedings of the 2005 Linux Symposium. Ottawa: [s.n.], 2005: 201-218.

[5] KHASHAN O A, ZIN A M, SUNDARARAJAN E A. ImgFS: a transparent cryptography for stored images using a filesystem in userspace[J]. Frontiers of Information Technology and Electronic Engineering, 2015, 16(1): 28-42.

[6] OUYANG X, RAJACHANDRASEKAR R, BESSERON X, et al. CRFS: a lightweight user-level filesystem for generic checkpoint/restart[C]// Proceedings of the 2011 International Conference on Parallel Processing. Piscataway: IEEE, 2011: 375-384.

[7] SZEREDI M. Fuse: filesystem in userspace[EB/OL]. [2020-01-25]. http://fuse.sourceforge.net.

[8] REN K, GIBSON G. TABLEFS: enhancing metadata efficiency in the local file system[C]// Proceedings of the 2013 USENIX Annual Technical Conference. Berkeley: USENIX, 2013: 145-156.

[9] HOLUPIREK A, GRÜN C, SCHOLL M H. BaseX & DeepFS joint storage for filesystem and database[C]// Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology. New York: ACM, 2009: 1108-1111.

[10] 薛矛，薛巍，舒继武，等. 一种云存储环境下的安全存储系统[J]. 计算机学报，2015，38(5)：987-998.(XUE M, XUE W, SHU J W, et al. A secure storage system over cloud storage environment[J]. Chinese Journal of Computers, 2015, 38(5): 987-998.)

[11] LEE G, SHIN S, SONG W, et al. Asynchronous I/O stack: a low-latency kernel I/O stack for ultra-low latency SSDs [C]// Proceedings of the 2019 USENIX Annual Technical Conference. Berkeley: USENIX, 2019: 603-616.

[12] YANG Z, HARRIS J R, WALKER B, et al. SPDK: a development kit to build high performance storage applications [C]// Proceedings of the 2017 IEEE International Conference on

Cloud Computing Technology and Science. Piscataway: IEEE, 2017: 154-161.

[13] YANG Z, LIU C, ZHOU Y, et al. SPDK vhost-NVMe: accelerating I/Os in virtual machines on NVMe SSDs via user space vhost target[C]// Proceedings of the IEEE 8th International Symposium on Cloud and Service Computing. Piscataway: IEEE, 2018: 67-76.

[14] LIU J, ARPACI-DUSSEAU A C, ARPACI-DUSSEAU R H, et al. File systems as processes [C/OL]// Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems. Berkeley: USENIX, 2019 [2020-01-11]. https://www.usenix.org/system/files/hotstorage19-paper-liu_0.pdf.

[15] ZHU Y, YU W, JIAO B, et al. Efficient user-level storage disaggregation for deep learning [C]// Proceedings of the 2019 IEEE International Conference on Cluster Computing. Piscataway: IEEE, 2019: 1-12.

[16] KALIA A, KAMINSKY M, ANDERSEN D G. Using RDMA efficiently for key-value services [C]// Proceedings of the 2014 ACM Conference on SIGCOMM. New York: ACM, 2014: 295-306.

[17] KAUR G, BALA M. RDMA over converged Ethernet: a review [J]. International Journal of Advances in Engineering and Technology, 2013, 6(4): 1890-1894.

[18] CAULFIELD A M, CHUNG E S, PUTNAM A, et al. A cloud-scale acceleration architecture [C]// Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture. Piscataway: IEEE, 2016:1-13.

[19] DRAGOJEVIĆ A, NARAYANAN D, NIGHTINGALE E B, et al. No compromises: distributed transactions with consistency, availability, and performance [C]// Proceedings of the 25th Symposium on Operating Systems Principles. New York: ACM, 2015: 54-70.

[20] WEI X, SHI J, CHEN Y, et al. Fast in-memory transaction processing using RDMA and HTM [C]// Proceedings of the 25th Symposium on Operating Systems Principles. New York: ACM, 2015: 87-104.

[21] ZAMANIAN E, BINNIG C, HARRIS T, et al. The end of a myth: Distributed transactions can scale [J]. Proceedings of the VLDB Endowment, 2017, 10(6): 685-696.

[22] 吴昊,陈康,武永卫,等. 基于RDMA和NVM的大数据系统一致性协议研究[J]. 大数据, 2019, 5(4): 89-99. (WU H, CHEN K, WU Y W, et al. Research on the consensus of big data systems based on RDMA and NVM[J]. Big Data Research, 2019, 5(4): 89-99.)

[23] 陈游旻,陆游游,罗圣美,等. 基于RDMA的分布式存储系统研究综述[J]. 计算机研究与发展, 2019, 56(2): 227-239. (CHEN Y M, LU Y Y, LUO S M, et al. Survey on RDMA-based distributed storage systems[J]. Journal of Computer Research and Development, 2019, 56(2): 227-239.)

[24] 董勇,周恩强,卢宇彤,等. 基于天河2高速互连网络实现混合层次文件系统H^{2}FS高速通信[J]. 计算机学报, 2017, 40(9): 1961-1979. (DONG Y, ZHOU E Q, LU Y T, et al. The implementation of communicating operation in hybrid hierarchy file system H^{2}FS with TH-Express 2 [J]. Chinese Journal of Computers, 2017, 40(9): 1961-1979.)

[25] DRAGOJEVIĆ A, NARAYANAN D, CASTRO M. RDMA reads: to use or not to use?[J]. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 2017, 40(1): 3-14.

[26] O'NEIL P, CHENG E, GAWLICK D, et al. The log-structured merge-tree (LSM-tree) [J]. Acta Informatica, 1996, 33(4): 351-385.

[27] GHEMAWAT S, DEAN J. LevelDB [EB/OL]. [2020-01-25]. https://github.com/google/leveldb.

[28] REN K, ZHENG Q, PATIL S, et al. IndexFS: scaling file system metadata performance with stateless caching and bulk insertion [C]// Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage and Analysis. Piscataway, IEEE, 2014: 237-248.

[29] XIAO L, REN K, ZHENG Q, et al. ShardFS vs. IndexFS: replication vs. caching strategies for distributed metadata management in cloud storage systems[C]// Proceedings of the 6th ACM Symposium on Cloud Computing. New York: ACM, 2015: 236-249.

[30] ZHENG Q, REN K, GIBSON G, et al. DeltaFS: exascale file systems scale better without dedicated servers[C]// Proceedings of the 10th Parallel Data Storage Workshop. New York: ACM, 2015: 1-6.

[31] LI S, LU Y, SHU J, et al. LocoFS: a loosely-coupled metadata service for distributed file systems[C]// Proceedings of the 2017 International Conference for High Performance Computing, Networking, Storage and Analysis. New York: ACM, 2017: No. 4.

[32] RPC: remote procedure call protocol specification version 2: RFC 1831[S]. Fremont, CA: Internet Engineering Task Force, 1995.

[33] SOUMAGNE J, KIMPE D, ZOUNMEVO J, et al. Mercury: enabling remote procedure call for high-performance computing [C]// Proceedings of the 2013 IEEE International Conference on Cluster Computing. Piscataway: IEEE, 2013: 1-8.

[34] OpenFabrics Interfaces Working Group. libfabric: Open Fabric Interfaces (OFI) [EB/OL]. [2020-01-25]. https://github.com/ofiwg/libfabric.

[35] BREITENFELD M S, FORTNER N, HENDERSON J, et al. DAOS for extreme-scale systems in scientific applications [EB/OL]. [2020-01-25]. https://arxiv.org/pdf/1712.00423.pdf.

[36] LOFSTEAD J, JIMENEZ I, MALTZAHN C, et al. DAOS and friends: a proposal for an exascale storage system [C]// Proceedings of the 2016 International Conference for High Performance Computing, Networking, Storage and Analysis. Piscataway: IEEE, 2016: 585-596.

[37] DHRUBA B. RocksDB: a persistent key-value store for flash and RAM storage: a library that provides an embeddable, persistent key-value store for fast storage [EB/OL]. [2020-01-25]. https://github.com/facebook/rocksdb.

[38] Storage Performance Development Kit. RocksDB: SPDK RocksDB mirror[EB/OL]. [2020-01-25]. https://github.com/spdk/rocksdb.

[39] MORRONE C, LOEWE B, MCLARTY T. mdtest HPC Benchmark [EB/OL]. [2020-01-25]. http://sourceforge.net/projects/mdtest.

This work is partially supported by the National Key Research and Development Program of China (2016YFB1000504).

DONG Haoyu, born in 1994, M. S. candidate. His research interests include file system, storage technology.

CHEN Kang, born in 1976, Ph. D., associate professor. His research interests include distributed system, system virtualization, machine learning.